

Reconstructing Twitter's Firehose

How to reconstruct over 99% of Twitter's firehose for any time period

Author: Jason Michael Baumgartner (Owner of Pushshift.io)

E-mail: jason@pushshift.io

Since there has been a lot of interest in detailing the methodology on how to do this, I've decided to make this a blog entry that will detail not only the technical aspects, but also to give some insight into the history of Twitter's ID creation scheme and some interesting tidbits on distributed computing in general. I hope that this article provides you with valuable insight into the many nuances of data science and data collection strategies. So without further ado, let's dive in!

On any random day, thousands of tweets are created every second on Twitter's social media platform. Twitter offers developers many different API endpoints that can be used to retrieve data, but there is no specific endpoint which allows reconstructing the thousands of tweets per second that are made to Twitter. Twitter does sell premium data services including their much coveted "firehose" stream. Twitter's firehose is a complete stream of all tweets made on their platform and is only available to a few businesses at an extraordinary price. This paper will detail methods on how to reconstruct portions of Twitter's firehose for research purposes.

The ability to ingest tweets for a specific time period can be invaluable for research. Whether your end goal is to gather a small sample of tweets between two time periods or ingest as much data from Twitter as possible, the technical methods covered in this paper will give you the ability to ingest various percentage levels of tweets (anywhere between one and 99% of all tweets between two time periods). The techniques outlined below will give you flexibility for ingesting a certain percentage of all publically available tweets.

Before diving into the technical details on how to implement this process, let's start by first rehashing some history. Twitter arrived on the social media scene in late March of 2006. During Twitter's infancy, there wasn't much data volume -- perhaps a couple of tweets every minute. At the beginning, Twitter started by using a sequential id scheme for its tweets and user ids. Since all of their ids were sequential, it was easy to transverse the space of possible tweets. Being sequential, all one had to do was simply request ids in a sequential fashion (1, 2, 3, 4, etc.). This worked out well during that time but also presented Twitter with one of its first problems -- the ability to create ids for

tweets and users in a distributed fashion without depending on one centralized service to assign ids. To better understand the problem, we first have to understand one of the main technical issues with a quickly growing platform such as Twitter.

Within a year of its launch, Twitter grew quickly and that rapid growth necessitated the need to add additional servers to handle the increasing load. During the time period when Twitter ids were sequential, there was one service responsible for generating and assigning ids for new tweets and users. This presented a single point of failure in their backend. Twitter increased their capacity to handle the growing volume of tweets and new users by adding more servers to process data. As more servers were added, their engineering team realized that they needed to decentralize id creation. If the service that handled id creation went offline, every server responsible for handling incoming tweets would be unable to assign a unique id to the tweet that server was processing. Their engineering team needed to find a decentralized and scalable solution for assigning ids to new incoming tweets.

There are a couple methods for assigning unique ids to new things in a decentralized way. Twitter could have elected to simply use a UUID scheme (https://en.wikipedia.org/wiki/Universally_unique_identifier) that would assign each new tweet a random hashed id. By using a large amount of bits for hash creation, the chance of possible id collisions would be virtually zero. However, Twitter wanted to find a way of assigning numeric ids to new objects in a way that didn't involve using hashes and also preserved some type of order thereby making ids sortable chronologically.

Enter the Snowflake



In 2010, Twitter engineers devised a method to decentralize id creation while also preserving order (maintaining the ability to sort tweets chronologically). This announcement was made on their blog detailing those efforts (https://blog.twitter.com/engineering/en_us/a/2010/announcing-snowflake.html). Basically, the method they came up with works like this:

By using the new Snowflake algorithm, each server would be able to generate a unique id on its own without the need to communicate with other servers. They did this by breaking up a tweet id into several components -- epoch millisecond timestamp, datacenter id, server id and sequence id. Each id created by the snowflake algorithm uses epoch time in milliseconds in the high order bits while also using the datacenter id, server id and sequence number in the low order bits.

Woah, wait a second -- high order, low order -- what? You've lost me.

(**Further reading:** https://en.wikipedia.org/wiki/Bit_numbering#Most_significant_bit)

I'll begin by giving a detailed explanation using examples to help clarify what all this means. Let's first talk about numbers and how machines see them. Any integer can be represented by bits in a computer (as long as you have enough bits to represent really large numbers). Let's take the integer 183. When we talk about integers, we usually always refer to them using base 10 math. However, computers do computation using base 2 mathematics -- more popularly known as binary.

Getting back to the number 183 -- this number can be expressed in binary as 10110111. In this example, it required 8 numbers in base 2 (binary) to represent a 3 digit number in base 10. So how is the number like 183 transformed to binary? How does 10110111 represent 183? Starting on the right side (low order), the first number is 1. A binary number is basically a bunch of switches where 1 represents "ON", 0 represents "OFF" and each slot represents a power of 2. The first far right number is 2 raised to the 0 (which is equal to one), the second number moving to the left is 2 raised to the 1 (which is 2), the next is 2 raised to the 2, which is four, etc. Basically, it looks like this (the top being the far right number and each one below it being the number to the left):

Binary Addition

1	=	2 ⁰	*	1	=	1
1	=	2 ¹	*	1	=	2
1	=	2 ²	*	1	=	4
0	=	2 ³	*	0	=	0
1	=	2 ⁴	*	1	=	16
1	=	2 ⁵	*	1	=	32
0	=	2 ⁶	*	0	=	0
1	=	2 ⁷	*	1	=	128

Adding up all the numbers on the far right (1,2,4,16,32,128) leads to 183. This is binary arithmetic. They say there are 10 types of people that understand binary -- and you are now in the group that does!

The digits to the right are considered low order bits and the ones towards the left are considered high order bits. In computer jargon, high order bits are also referred to as MSB (most significant bits) and these numbers represent the higher values. Great, so how does this relate to Twitter ids?

Remember, Twitter had a couple major goals when creating their Snowflake algorithm -- every server could create ids without conflicting with another server and all ids created would preserve some type of ordering (in this case, temporal). Let's look at an actual Twitter ID and show which parts represent what. I'll use a tweet id from one of my tweets: 1100125195476631553. I will use Python commands to help illustrate examples and give you the opportunity to try them out for yourself. This id contains all the information needed to tell you when it was created down to the millisecond and which datacenter and server created it, and what sequence id value it represents. In order to do this, we first have to convert it to its binary equivalent (brace yourself):

```
>>> bin(1100125195476631553)
```

```
1111010001000110111000001001010111001101011110000000000000001
```

Twitter stores the following information in the binary representation of an id:

The first 12 digits (least significant or LSB bits -- the numbers furthest to the right) represent the sequence id. In this example, the sequence id is '000000000001' or simply 1.

The next 5 binary numbers contain the server id for the tweet.
'11000' = 24

The next 5 binary numbers contain the data center id for the tweet:
'01011' = 11

The next 22 binary numbers contain the time in milliseconds of the tweet. However, there is a small catch -- there is an offset that needs to be applied. The offset value used is when Snowflake officially started. That offset is 1288834974657. So in order to

get the time in milliseconds of when the tweet was made, you would take the value of the next 22 binary digits and add it to this offset.

'11110100010001101110000010010101110011' = 262290285939 + 1288834974657 = 1551125260596

The epoch time of creation for this tweet is 1551125260.596 or (UTC) Monday, February 25, 2019 8:07:40 PM (and .596 milliseconds)

Here is the tweet ID again with each part separated out and labeled:

Time Component	DC	SID	Sequence ID
11110100010001101110000010010101110011	01011	11000	000000000001

DC = Datacenter id
SID = Server id

Hopefully this removes some of the mystery behind Twitter tweet ids and gives you a better understanding of how they are created. The reason the time component is located in the higher order bits is to keep the ids sortable by time. By having the time component in the most significant bits, temporal order is preserved and tweet ids can be sorted chronologically. It's important that you understand how Twitter creates their tweet ids before moving on -- I encourage you to reread this section if this still feels a bit unclear.

Using Snowflake to our advantage

Understanding how tweet ids are created is the first step in helping to reduce the id space that needs to be scanned in order to find actual tweets for any given time range.

For each millisecond in the timeline, how many tweet ids are possible? We know that the first 22 bits contain the sequence id, server id and data center id. If we knew nothing about Twitter's infrastructure, we'd have to check every combination. How many combinations are there with 22 bits? There are a total of 4,194,304 possible values that can be represented with 22 bits! Let's do some quick math to see if it is feasible to gather tweets by scanning such a large space.

[Twitter's statuses lookup API endpoint](#) allows for a total of 1,200 API calls every 15 minutes. Each call allows the user to pass 100 ids for a total of 120,000 id requests every 15 minutes using both APP auth (300 requests) and USER auth (900 requests).

This would require making API calls over 525 minutes to scan every possible value -- and that's just to check one millisecond of actual timeline data! It is absolutely unfeasible to do this given those API limits. But can we do better? If you're reading this, you know we can!

(**Note:** For the remainder of this article, I will treat the bit space representing the server id and data center id as one entity by combining them. Realistically, we aren't concerned with what data center was used, we mainly care about combinations and permutations. This is best done by treating these two different ids conceptually as one "machine id." Recall that 5 bits are used for the data center id and 5 bits are used for the server id -- so if we combine them both, we'll treat those ten bits as one entity and refer to this as the machine id.)

Taking advantage of Twitter's Snowflake scheme

When I ran an analysis on millions of tweets, I made a surprising observation -- Twitter doesn't use that many machines to create their ids. In fact, the bulk of ids are created by 20 or fewer machines. This reduces the space tremendously. Now we only have $20 \times$ the number of possible sequence ids that can be represented by 12 bits (remember, the sequence id is the first 12 bits). How many possible values are there in 12 bits? There are 4,096 possible values to check. We need to check all 4,096 for each machine (20). That means the total space is now just 81,920 instead of 4,194,304. We've reduced the id space that we have to check by a factor of over 50!

Unfortunately, there is still a problem -- it will still require over 10 minutes to scan that space for each millisecond of time. In order to scan an entire second of actual history, it will require over 10,000 minutes. That means it will take a week of constant API calls just to reconstruct one second of Twitter firehose data. We've gone from the realm of impossible to absurd -- but can we do better? In fact, we can!

After analyzing millions of tweets to see what sequence ids are most used, I found that most tweets only use a sequence ID of 0! The next largest batch of tweets uses a sequence ID of 1. In fact, over 99% of all tweets use a sequence id of 10 or less! Here is a breakdown of sequence id usage when I checked one million tweets:

Seq ID | % of Tweets | Cumulative %

The main sequence ids

0	0.49815	0.49815
1	0.26266	0.76080
2	0.12046	0.88126
6	0.04343	0.92469

5	0.02783	0.95252
3	0.02588	0.97839
7	0.00747	0.98586
4	0.00735	0.99321 ← over 99%!
8	0.00304	0.99625
10	0.00109	0.99734

What this shows is that almost half of all tweets have a sequence id of 0. Over 75% of all tweets have a sequence id of 0 or 1. Over 90% of all tweets have a sequence id of 0,1,2 or 6. (These numbers may change with a larger sample -- I suspect that sequence ids are handed out sequentially -- although I have seen very high sequence ids in the past so there may be more fine-tuning possible).

This is amazing because this means that, theoretically, you could get almost half of all available tweets by just checking tweets with a sequence id of 0! How does this reduce the space of ids that we need to check?

20 machine ids * 1 (one sequence id) = 20 total id scans for each millisecond of time. With 120,000 possible id requests every 15 minutes (using one dev account), we can scan 6,000 milliseconds of actual timeline data (6 seconds). That doesn't sound like a lot, but that is equal to 576 seconds of timeline data each day -- almost ten minutes of timeline data getting close to 50% of all tweets! We've gone from impossible to absurd to manageable!

For many projects, you wouldn't need to collect a 50% sample. What if we wanted a 10% sample for a specific range of time? Let's take a look and see what percentage of tweets are processed by each machine id:

Machine ID | % of Tweets | Cumulative %

The first 20 machine ids

332	0.09057	0.09057
335	0.05927	0.14984
361	0.05724	0.20707
363	0.05452	0.26159
381	0.05406	0.31565
364	0.05378	0.36944
382	0.05344	0.42288
372	0.05341	0.47629
362	0.05263	0.52892

376	0.05247	0.58139	
375	0.05174	0.63312	
350	0.05145	0.68458	
365	0.04923	0.73380	
325	0.04076	0.77456	
347	0.03860	0.81317	
326	0.03855	0.85172	
336	0.03748	0.88920	
333	0.03660	0.92581	
327	0.03576	0.96157	
342	0.03453	0.99610	← over 99%!

For the time range I sampled, 10% of tweets were handled by machine id 332. 15% were handled by 2 machine ids. 20% were handled by 3 machine ids. In fact, over 99% of all tweets were handled by twenty machine ids in total. If we wanted a 10% sample, let's assume we would need to scan the first five machine ids. How does that reduce the space of ids that we need to check?

120,000 id checks against 5 machine ids with sequence 0 should give us around a 15% sample (remember, around 49% of all tweets have a sequence id of 0). That means we can now sample four times faster if we want a ~ 15% sample. We're now approaching 35 minutes of timeline data for each day of processing (using one dev account)!

If ~99% of all tweets were desired, it would require 200 id lookups per millisecond of timeline data. Using the maximum number of requests for one developer account, it would be possible to scan 6 seconds of timeline data every 150 minutes (a little less than one minute of timeline data per day of requests). If multiple developer accounts were used, or an application asked for user authorization to scan more ids (900 additional requests per 15 minute window for each user that authorized the app), it could be feasible to reconstruct large portions of the timeline by making parallel requests and cycling through user keys. An application that collected 100 user authorizations could scan close to 100 minutes of timeline data per day. Taken to an extreme, if an application had ~1,650 users authorizing the app to make requests, the entire firehose itself (~99% sample rate) could be reconstructed in real-time. Such a system could be designed to make requests from multiple servers and guided by a master control program to reconstruct Twitter's entire firehose!

(Author's note: It is my belief that if enough university students and researchers worked together, we could reproduce the firehose at a 99+% sample rate and make the data available to all researchers -- that would be amazingly cool! Getting ~1,700 people (a few extra as a buffer for incomplete calls, etc.) to participate wouldn't be an insurmountable task and if there is interest in doing this project, I would be

happy to help write the code to manage everything including the distribution of the stream to various universities, etc. We would only need to get around 1,700 users to authorize the app to make status lookups on their behalf -- the application would not require any sensitive permissions. My e-mail is jason@pushshift.io if you are interested in kick-starting such an adventure. Creating a “decahose” stream (~10%) would only require around 50-75 users to participate! A 50% stream would require a few hundred people.)

Keep in mind that while ingesting this data, you will get retweets that contain the original tweet. That means the likelihood of getting tweets that were popular during the time period approaches 100% the more often it was retweeted.

Doing an initial pre-scan to determine what machine ids were used

One important step in this process is doing an initial pre-scan for the time period you are interested in ingesting. The purpose of the pre-scan is to determine which machine ids were in use during a specific time period. Over time, Twitter has added and removed machines, so the machine ids do change over time. The pre-scan should cycle over all possible ten bit variables that make up the machine id (5 bits are reserved for the server id and 5 bits are reserved for the data-center id). There are 1,024 possible machine ids, so the pre-scan will need to iterate over all possible machine ids for the time range that will be ingested. This prescan does not have to collect a large amount of tweets, but the pre-scan should use at a minimum 1,200 API calls (15 minutes worth of API calls for the statuses lookup endpoint) to get all the machine ids that were in use during the timespan that will be ingested.

1,200 API calls will allow for 120,000 id checks. With this amount, 117 milliseconds of time can be scanned checking all possible machine ids.

(**Note:** I will add this functionality to the firehose ingest script and will call the method “prescan.” The prescan method will accept a start period, end period and number of checks. The function will then space out the requests between the start period and end period to determine which machine ids were in play during the time range. I will include this function in the eventual Github commit for the firehose reconstruction code.)

Statistical Probability of ingesting a tweet retweeted N times

(**Edit note:** Add some examples / chart of the probability of ingesting a tweet that was retweeted given a specific sampling percentage vs. how many times a tweet was retweeted.)

For example, if one ingests a 10% random sample and a tweet is retweeted 10 times, the probability of ingesting that tweet is $(1 - (.90^{10})) = \sim 65\%$ chance of getting that tweet.

Generalized where X is the sample rate percentage and N is the number of times something was retweeted:

$$(1 - (1.00 - X^N))$$

Example:

X = .25 (25% Random sample stream)

N = 5 (Retweeted at least 5 times)

$(1 - (1.00 - .25)^5) = 76.26\%$ chance of ingesting the original tweet

Working backwards -- To have a 95% probability of ingesting something from a 25% stream, you would need it to occur at least $\log(0.05)/\log(1 - 0.25) = 10.4$ times.

(Author note: A big thank you to Xanda Schofield Twitter: @XandaSchofield for help with the math.)

Use Cases:

Temporal Analysis of Significant Events

A significant event happens and a researcher wishes to find the first tweet mentioning an earthquake, terrorist attack, etc. -- Using this method, one could use the Twitter search API but the public search endpoint only has ~10 day history. The method outlined in this paper would allow researchers to check historical events to find the first tweets that mention something. (Could be awesome for Earthquakes and getting a 5 minute window of tweets to get every tweet that mentions feeling the ground shake, etc.)

Complete sample to better understand SPAM / BOT Retweet activity

President Trump makes a tweet -- how was his tweet retweeted during the first few minutes afterwards? This method would help to detect bot-like activity on Twitter by scanning time blocks right after a specific account makes a tweet to see if the same accounts are active each time.

Creating a real-time stream

One could create self-adapting code that utilizes all available authorizations to stream the firehose at X% (depending on the number of auth accounts in the rotation pool). With one dev account, it should be possible to get a real-time stream with around 1% of all tweets that could easily scale up as more authorizations are added to the pool. If approximately 50-75 users gave the app authorization to use the statuses/lookup endpoint, a real-time decahose (10%) would be possible.

Actual Code Examples

How to break apart a Twitter ID and view its components using Python:

(Edit Note: Add this code to Github)

```
#!/usr/bin/env python3

# This small script shows how to deconstruct a Twitter tweet id into its
# various components. The tweet_components method accepts a tweet id and
# returns a dict object with key / values representing the various
# components of a tweet id. Each component has its own method detailing
# how values are extracted from the tweet id.

def sequence_id(id):
    # Return the first 12 bits using a mask
    # the sequence id is composed of the first 12 bits
    return id & 0b111111111111

def machine_id(id):
    # right bitshift 12 and apply an AND mask to get 10 rightmost bits
    # this is a combination of server id and datacenter id
    return (id >> 12) & 0b1111111111

def server_id(id):
    # right bitshift 12 and apply an AND mask to get the next 5 rightmost bits
    # the server id is composed of bits 13-17 (starting from the right)
    return (id >> 12) & 0b11111

def datacenter_id(id):
    # right bitshift 17 and apply an AND mask to get the next 5 rightmost bits
    # the datacenter id is composed of bits 18-22 (starting from the right)
    return (id >> 17) & 0b11111

def creation_time(id):
    # right bitshift 22 and apply Snowflake offset
```

```

    # the epoch time (in milliseconds) are the first 22 MSB bits (first 22
bits starting from the left)
    return ((id >> 22) + 1288834974657)

def tweet_components(tweet_id):
    tweet_id = int(tweet_id) # Convert to int if str is accidentally passed in
    c = {} # Components
    c['sequence_id'] = sequence_id(tweet_id)
    c['machine_id'] = machine_id(tweet_id)
    c['server_id'] = server_id(tweet_id)
    c['datacenter_id'] = datacenter_id(tweet_id)
    c['creation_time_milli'] = creation_time(tweet_id)
    return(c)

tweet_id = 1100125195476631553
data = tweet_components(tweet_id)
print(data)

>>> {'sequence_id': 1, 'machine_id': 376, 'server_id': 24, 'datacenter_id':
11, 'creation_time_milli': 1551125260596}

```

How to efficiently scan Twitter's tweet id space for a specific time range:

(Edit Note: Commit code to Github and provide link here -- I will be refactoring the code below to work without using Tweepy and simplify a few things so that people can start testing it immediately)

.

SUPER ROUGH DRAFT CODE BELOW:

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import tweepy
from tweepy import OAuthHandler
import ujson as json
import time
import configparser
from collections import deque, defaultdict
import sys
import os

class firehose:

    # Read Credentials
    Config = configparser.ConfigParser()
    Config.read("credentials.ini")
    consumer_key = Config.get("TwitterCredentials", "consumer_key")
    consumer_secret = Config.get("TwitterCredentials", "consumer_secret")
    access_token = Config.get("TwitterCredentials", "access_token")

```

```

    access_token_secret =
Config.get("TwitterCredentials", "access_token_secret")

    # This will cycle between APP and USER authentication for one dev account.
Multiple accounts can be added

    api = []
    auth = tweepy.OAuthHandler(consumer_key, consumer_secret)

api.append(tweepy.API(auth, wait_on_rate_limit=True, wait_on_rate_limit_notify=T
rue, compression=True))

    auth_b = tweepy.OAuthHandler(consumer_key, consumer_secret)
    auth_b.set_access_token(access_token, access_token_secret)

api.append(tweepy.API(auth_b, wait_on_rate_limit=True, wait_on_rate_limit_notify
=True, compression=True))

    current_api = 0
    api_counter = 0
    failures = 0

    MACHINE_IDS =
(375, 382, 361, 372, 364, 381, 376, 365, 363, 362, 350, 325, 335, 333, 342, 326, 327, 336, 347, 3
32)
    SNOWFLAKE_EPOCH = 1288834974657

    def __init__(self):
        self.queue = deque()
        self.fh = open("firehose_test.ndjson", "a+")
        self.ratelimit_reset = None
        self.ratelimit_remaining = None

    def get_creation_time(self, id):
        return ((id >> 22) + 1288834974657)

    def machine_id(self, id):
        return (id >> 12) & 0b1111111111

    def sequence_id(self, id):
        return id & 0b111111111111

    def ingest_range(self, begin, end): # This method is where the magic happens
        for epoch in range(begin, end): # Move through each millisecond
            time_component = (epoch - self.SNOWFLAKE_EPOCH) << 22
            for machine_id in self.MACHINE_IDS: # Iterate over machine ids
                for sequence_id in [0]: # Add more sequence ids as needed

```

```

sequence_id

        twitter_id = time_component + (machine_id << 12) +
sequence_id

        self.queue.append(twitter_id)
        if len(self.queue) >= 100:
            ids_to_process = []
            for i in range(0,100):
                ids_to_process.append(self.queue.popleft())
            self.process_ids(ids_to_process)

    def process_ids(self, tweet_ids):
        tweets =
        firehose.api[firehose.current_api].statuses_lookup(tweet_ids, tweet_mode='extended', trim_user=False, include_entities=True)
        if 'x-rate-limit-remaining' in
        firehose.api[firehose.current_api].last_response.headers:
            self.ratelimit_remaining =
        int(firehose.api[firehose.current_api].last_response.headers['x-rate-limit-remaining'])
        if 'x-rate-limit-reset' in
        firehose.api[firehose.current_api].last_response.headers:
            self.ratelimit_reset =
        int(firehose.api[firehose.current_api].last_response.headers['x-rate-limit-reset'])

        tweets_processed = defaultdict(int)
        for tweet in tweets:
            tweet._json['retrieved_on'] = int(time.time())

        print(json.dumps(tweet._json, sort_keys=True, ensure_ascii=True), file=self.fh)
            created_at = tweet._json['created_at']
            id = int(tweet._json['id'])

        print(self.machine_id(id), self.get_creation_time(id), self.sequence_id(id))
            tweets_processed[self.get_creation_time(id)] += 1

        print(self.ratelimit_remaining)
        if self.ratelimit_remaining <= 0:
            firehose.api_counter += 1
            firehose.current_api = firehose.api_counter % len(firehose.api)

        fh = firehose()

    while True:
        try:
            start = int(time.time() * 1000) - 1000 # Start from current time
            end = start + 5000 # Get five seconds of the timeline
            fh.ingest_range(start, end)
        except Exception as e:

```

```
print(e)
time.sleep(60)
```

(**Author Note:** I will commit better code soon to Github -- this should get most people started for now. I should have code examples committed to Github by this weekend (March 2-3) with instructions to get rolling quickly.)